**//mapTest.cpp Please fill up every case down there.**

/\* These are test for map inerface methods using Google Test \*/
#include "map.h"
#include <gtest/gtest.h>
#include <iostream>
using namespace std;

//The constructor should create a map with zero size
TEST(MapTest, constructorCreatesZeroSizeMap)
{
}

//Inserting into an empty map should return true and result in a size of one
TEST(MapTest, insertWhileEmptyReturnsTrueAndMapSizeIsOne)
{
}

//Inserting a key that already exists should fail and not change the map size
TEST(MapTest, insertOfSameKeyReturnsFalseAndMapSizeRemainsSame)
{
}

//Accessing an existing item using [] should return correct value and not
//change the map size
TEST(MapTest, indexOperatorOfExistingKeyReturnsProperValueAndSizeIsSame)
{
}

//Using [] to set an item in an empty map should add the item with an
//appropriate value and increase the map size to one
TEST(MapTest, indexOperatorOnEmptyMapProperlySetsValueAndSizeIsOne)
{
}

//Using [] to set an item in a map that already contains other items
//inserts and updates the size appropriately
TEST(MapTest, indexOperatorOnMissingKeyProperlySetsValueAndIncrementsSize)
{
}

//Erasing a key that exists in the map succeeds and decrements the map size
TEST(MapTest, eraseOfExistingKeyReturnsTrueAndDecrementsSize)
{
}

//Erase on an empty map fails and keeps the map size at zero
TEST(MapTest, eraseOnEmptyMapReturnsFalseAndSizeRemainsZero)
{
}

//Erase of a non-existant key fails and does not change the map size
TEST(MapTest, eraseOfMissingKeyReturnsFalseAndSizeRemainsSame)
{
}

//Using the copy constructor results in two maps of the same, correct size
TEST(MapTest, copyConstructorMakesCorrectSize)
{
}

//Using the copy constructor with an existing map makes its own
//copy, such that if the value associated with a key is changed
//in the copy it does not change the associated value in the
//original, and inserts/erases on either won't affect the other.
TEST(MapTest, copyConstructorMakesSeparateCopy)
{
}

//Using the assignment operator results in two maps of the same, correct size
TEST(MapTest, assignmentOperatorMakesCorrectSize)
{
}

//Using the assignment operator with an existing map makes its own
//copy, such that if the value associated with a key is changed
//in the copy it does not change the associated value in the
//original, and inserts/erases on either won't affect the other.
TEST(MapTest, assignmentMakesSeparateCopy)
{
}

//Using the assignment operator with an existing map with over 1000 items
//makes a copy that has correct keys/values at a couple of places deep within
//the map (values that aren't at either end of the range of keys used).
TEST(MapTest, assignmentWithManyItemsHasCorrectValues)
{
}

int main(int argc, char \*\*argv) {
 ::testing::InitGoogleTest(&argc, argv);
 return RUN\_ALL\_TESTS();

}

**//The header file map.h to be used above is given below. Don’t make any changes here, for reference.**

// A non-generic Map ADT implemented with a BST data structure
// The BST is not-balanced and non-threaded

#ifndef MAP\_H
#define MAP\_H
#include <iostream>
#include <string>

using namespace std;

typedef string KEY\_TYPE;
typedef string VALUE\_TYPE;

class Map{
 struct Elem; //declaration of an interal structure needed below...

 public:
 //---Constructors and destructors---
 Map(); // constructs empty Map
 Map(const Map &rhs); // copy constructor
 ~Map(); // destructor

 // assignment operator
 Map& operator=(const Map &rhs);

 // insert an element; return true if successful
 bool insert(KEY\_TYPE, VALUE\_TYPE);

 // remove an element; return true if successful
 bool erase(KEY\_TYPE);

 // return size of the Map
 int size() const;

 // return an iterator pointing to the end if an element is not found,
 // otherwise, return an iterator to the element
 class Iterator;
 Iterator find(KEY\_TYPE) const;

 // Iterators for accessing beginning and end of collection
 Iterator begin() const;
 Iterator end() const;

 // overloaded subscript operator
 VALUE\_TYPE& operator[](KEY\_TYPE);

 // output the undering BST
 ostream& dump(ostream& out) const;

 // a simple Iterator, won't traverse the collection
 class Iterator{
 public:
 Iterator(){}
 explicit Iterator(Elem \*cur):\_cur(cur) {}
 Elem& operator\*();
 Elem\* operator->();
 // Iterator operator++(int);
 bool operator==(Iterator it);
 bool operator!=(Iterator it);
 private:
 Elem\* \_cur;
 };

private:
 struct Elem {
 KEY\_TYPE key;
 VALUE\_TYPE data;
 Elem \*left;
 Elem \*right;
 };
 Elem \*\_root; // a dummy root sentinel
 int \_size;

 // helper method for inserting record into tree.
 bool insert(Elem \*& root, const KEY\_TYPE& key, const VALUE\_TYPE& data);

 // helper method for print tree
 void printTree(ostream& out, int level, Elem \*p) const;

 // common code for deallocation
 void destructCode(Elem \*& p);

 // common code for copy tree
 void copyCode(Elem\* &newRoot, Elem\* origRoot);
};

ostream& operator<< (ostream&, const Map&);

#endif